Towards Gradual Typing for Generics

Lintaro Ina and Atsushi Igarashi

Graduate School of Informatics, Kyoto University
{ina,igarashi}@kuis.kyoto-u.ac.jp

Abstract. Gradual typing, proposed by Siek and Taha, is a framework to combine the benefits of static and dynamic typing. Under gradual typing, some parts of the program are type-checked at compile time, and the other parts are type-checked at run time. The main advantage of gradual typing is that a programmer can write a program rapidly without static type annotations in the beginning of development, then add type annotations as the development progresses and end up with a fully statically typed program; and all these development steps are carried out in a single language.
This paper reports work in progress on the introduction of gradual typing into class-based object-oriented programming languages with generics. In previous work, we have developed a gradual typing system for Featherweight Java and proved that statically typed parts do not go wrong. After reviewing the previous work, we discuss issues raised when generics are introduced, and sketch a formalization of our solutions.

1 Introduction

Siek and Taha have coined the term "gradual typing" [1] for a linguistic support of the evolution from dynamically typed code, which is suitable for rapid prototyping, to fully statically typed code, which enjoys type safety properties, in a *single* programming language. The main technical challenge is to ensure some safety property even for *partially typed* programs, in which some part is statically typed and the rest is dynamically typed.

The framework of gradual typing consists of two languages: the surface language in which programmers write programs and the intermediate language into which the surface language translates. The type system of the surface language allows dynamically and statically typed portions to be seamlessly mixed by, for example, allowing an expression of dynamic type to be passed to a statically typed parameter. In order to ensure safety of statically typed parts, run-time checks are required on the "border" between the statically typed and dynamically typed worlds. The translation into the intermediate language makes these run-time checks explicit. The expected safety property then would be that a well-typed surface language program translates to a well-typed intermediate language program, which can fail only at run-time checks inserted by the translation. When the program is fully statically typed, no checks will be inserted, hence no run-time type errors will occur.

In this paper, we discuss our work in progress on the theory of gradual typing for class-based object-oriented programming languages with generics, such as Java and C#. C# 4.0 [2] is going to include dynamic types but it has not been mentioned how dynamic types and generics are combined. Although Siek and Taha [3] have studied gradual typing for Abadi-Cardelli's first-order object-calculus with subtyping [4], (as far as we know) there have been no study on gradual typing for a class-based language with a nominal type system and generics.

We first report on our previous work [5] on gradual typing for Featherweight Java (FJ) [6]. As in Siek and Taha [1, 3], we have added the dynamic type `?`, a (static) type identifier for dynamically typed expressions. Since FJ is a simplest possible choice as the base language, actually, there is little additional technical subtlety compared to other settings. Nevertheless, we believe it is interesting to review, since the formal translation shows how gradual typing can be implemented with the combination of typecasts and reflection. We also point out another safety property of gradual typing: "potentially typeable programs do not go wrong", which is proved for gradually typed FJ. Intuitively, it means that, if a given program can be made statically typed only by adding static type annotations, then it can already run safely. In other words, the inserted run-time checks would not prevent safe execution, which would have been ensured only by adding static type annotations.

Then, we discuss issues in the combination of generics and gradual typing. One natural consequence of this combination is that type variables may be instantiated with `?`. To deal with such an instantiation in the presence of bounded polymorphism, we introduce the notion of *bounded dynamic types*, which have characteristics of both `?` and ordinary types. Another consequence, which we discuss, is that the run-time type of an object may involve `?`. Finally, we sketch the highlights of our formalization work, which is in the middle of the development.

Related Work. There is much work on mixing dynamic and static types (see, for example, Siek and Taha [3] for a more extensive survey). Here, we compare our work mainly with related work on object-oriented programming languages.

Some type systems are proposed to achieve benefits of static type checking in dynamically typed languages. Bracha and Griswold [7] have proposed *Strongtalk*, which is a typechecker for a downward compatible variant of Smalltalk, a dynamically typed class-based object-oriented programming language. The type system is structural and supports subtyping and generics. *Strongtalk* does not accept partially typed programs. Thiemann [8] have proposed a type system for (a subset of) JavaScript, which is a prototype-based object-oriented language, to avoid some kind of run-time errors by static type checking. This work does not attempt to support the evolution from dynamically typed program to statically typed program. Furr, An, Foster, and Hicks [9] have developed Diamondback Ruby, an extension of Ruby with a static type system. Their type system, which seems useful to find bugs, however, does not offer static type safety.

Anderson and Drossopoulou [10] have proposed a type system for (a subset of) JavaScript for the evolution from JavaScript to Java. Although it is nominal and concerned about script-to-program evolution, their type system does not have subtyping, inheritance, or polymorphism; moreover, this work is not concerned about safety of partially typed programs in the middle of the evolution.

Lagorio and Zucca [11] have developed Just, an extension of Java with unknown types. Although there is some overlap in the expected uses of this system and gradual typing, the main purpose of unknown types is to omit type declarations; possibly unsafe use of unknown types is rejected by the type system. They use reflection to implement member access on unknown types.

Gray, Findler, and Flatt [12] have implemented an extension of Java with dynamic types and contract checking [13] for interoperability with Scheme. However, they mainly focus on the design and implementation issues and give no discussion on the interaction with generics. Their technique to implement reflective calls can be used for our setting.

As we have already mentioned, Siek and Taha have studied gradual typing for Abadi-Cardelli's object calculus [3]. However, the language is object-based and parametric polymorphism is not studied. Another point is that the implementation of run-time checks for class-based languages seems easier than that for object-based languages, since, in class-based languages, every value is tagged with its run-time type information and the check can be performed in one step (unlike higher-order contract checking, which checks inputs to and outputs from functions separately).

Siek and Vachharajani [14] and Matthews and Ahmed [15] have discussed the combination of dynamic typing and parametric polymorphism in the λ-calculus. Aside from the difference in target languages, they have not addressed *bounded* polymorphism.

The Rest of This Paper. In Section 2, we review our previous work on gradual typing for Featherweight Java. Then, in Section 3, we discuss issues in extending it to generics. Section 4 sketches our formalization work and Section 5 gives concluding remarks. Throughout the paper, we assume basic knowledges about the formalism of Featherweight Java [6].

2 Summary of Gradual Typing for FJ

In this section, we review the previous work [5] on gradual typing for Featherweight Java (FJ) [6]. Just as other work on gradual typing, there are two languages: the surface language called $\mathrm{FJ}^{?}$, which is equipped with the dynamic type `?`;[1] and the intermediate language called $\mathrm{FJ}_{\mathrm{refl}}$, which has reflective member access operations. The relation between the two languages is illustrated in Figure 1 with some of the judgment forms in these languages. Although $\mathrm{FJ}^{?}$ has a type system, its semantics is given only by a translation, denoted by $\rightsquigarrow$, to make run-time checks explicit. $\mathrm{FJ}_{\mathrm{refl}}$ is given both a type system and a direct operational semantics.

[1] Note that `?` is already reserved in Java proper for wildcard types [16]; here, we follow the notational convention used in other work on gradual typing.

$\mathrm{FJ}^? = \mathrm{FJ} + \text{dynamic type}$	translation	$\mathrm{FJ}_{\mathrm{refl}} = \mathrm{FJ} + \text{reflection}$
typing ($\Gamma \vdash_{\mathrm{G}} \mathtt{e}_g : \mathtt{T}$)	$\rightsquigarrow$	typing ($\Gamma \vdash_{\mathrm{R}} \mathtt{e}_r : \mathtt{T}$)
	($\Gamma \vdash \mathtt{e}_g \rightsquigarrow \mathtt{e}_r : \mathtt{T}$)	reduction ($\mathtt{e}_r \longrightarrow_{\mathrm{R}} \mathtt{e}'_r$)

Fig. 1. $\mathrm{FJ}^?$ and $\mathrm{FJ}_{\mathrm{refl}}$.

We first describe typing in $\mathrm{FJ}^?$ and translation to $\mathrm{FJ}_{\mathrm{refl}}$ informally by means of a small example, and then give an excerpt from the formalization and the theorems of safety properties.

2.1 Informal Review

In $\mathrm{FJ}^?$, `?` can be used, instead of class names, where types are expected; and it means the expression is dynamically typed. For example, in the following class table, class `W` has field `f` of dynamic type. (◁ is an abbreviation for `extends`.)

```
class X ◁ Object {                    class A ◁ Object { Object f; }
  Object m(A x) { return x.f; }       class B ◁ Object { }
}                                     class W ◁ Object { ? f; }
```

`?` resembles `Object` in that a variable of type `?` can be bound to any object; however, it is more permissible than `Object`. First, an expression of type `?` can be passed to anywhere. Consider the following example.

```
new X().m(new B());             // rejected
new X().m(new W(new B()).f);    // well typed
```

The first expression will be rejected in type checking because it attempts to pass the expression of type `B` as an argument of type `A`. The second expression, on the other hand, is well typed since the static type of the field access is `?`, even though `new B()` will be eventually passed through to the method `m`. Second, any method invocation or field access on `?` is allowed and the whole expression is, again, given type `?`, while `Object` (in FJ) does not allow any.

The semantics of $\mathrm{FJ}^?$ is given by translation into $\mathrm{FJ}_{\mathrm{refl}}$, in which run-time checks are explicit. Since $\mathrm{FJ}_{\mathrm{refl}}$ does not have dynamic types, all occurrences of `?` will be simply replaced by `Object`. When an expression of type `?` is passed to a typed parameter, a run-time typecheck is inserted to prevent erroneous values from flowing into a typed context. Method invocations and field accesses whose receiver is of type `?` in $\mathrm{FJ}^?$ are translated into reflective member accesses, written `invoke()` and `get()`, respectively, of $\mathrm{FJ}_{\mathrm{refl}}$. The following example shows how this goes.

$$
\begin{array}{ll}
\mathtt{new\,X().m(new\,W(new\,B()).f)} & \rightsquigarrow \mathtt{new\,X().m(\langle\!\langle A\rangle\!\rangle new\,W(new\,B()).f)} \\
 & \rightarrow \mathtt{new\,X().m(}\underline{\mathtt{\langle\!\langle A\rangle\!\rangle new\,B()}}\mathtt{)} \\
\mathtt{new\,W(new\,X()).f.m(new\,B())} & \rightsquigarrow \mathtt{invoke(new\,W(new\,X()).f,m,new\,B())} \\
 & \rightarrow \mathtt{invoke(new\,X(),m,new\,B())} \\
 & \rightarrow \underline{\mathtt{(\langle\!\langle A\rangle\!\rangle new\,B()}}\mathtt{).f}
\end{array}
$$

Here, $\rightsquigarrow$ denotes the translation from $\mathrm{FJ}^?$ into $\mathrm{FJ}_{\mathrm{refl}}$; $\rightarrow$ denotes a reduction step in $\mathrm{FJ}_{\mathrm{refl}}$; and $\llparenthesis \mathtt{C} \rrparenthesis$ denotes a run-time typecheck inserted by the translation—actually, its semantics is the same as ordinary typecasts in this setting, so we use the ordinary notation `(C)` in what follows in this section. $\mathtt{invoke(e,m,}\overline{\mathtt{e}}\mathtt{)}$ invokes method `m` of (the value of) `e`; it is checked whether the method exists, whether the number of arguments agree, and whether the (run-time) type of the actual argument $\mathtt{e}_i$ is a subtype of the formal argument type. In the translation/reduction sequences above, the underlined parts denote run-time errors. The first one shows that an invalid argument is detected before `m` has been invoked and the second shows that an invalid argument is detected at a reflective method invocation.[2]

As we will see below, we have proved that an $\mathrm{FJ}_{\mathrm{refl}}$ program translated from a well-typed $\mathrm{FJ}^?$ program is also well typed and a well-typed $\mathrm{FJ}_{\mathrm{refl}}$ program never yields run-time errors except one caused by reflective member accesses or run-time typechecks inserted by the translation.

2.2 Formalization

As we have mentioned earlier, the syntax of $\mathrm{FJ}^?$ is the same as FJ except that we can use `?` as a type. We show the two typing rules for method invocations, which are most interesting. The judgment $\Gamma \vdash_{\mathrm{G}} \mathtt{e} : \mathtt{T}$ means that expression `e` is of type `T` in the environment Γ (`T`, `U`, and `V` range over types).

Definition 1 (Typing).

$$\frac{\begin{array}{cc}\Gamma \vdash_{\mathrm{G}} \mathtt{e}_0 : \mathtt{C}_0 & \Gamma \vdash_{\mathrm{G}} \overline{\mathtt{e}} : \overline{\mathtt{U}} \\ \mathit{mtype}(\mathtt{m}, \mathtt{C}_0) = \overline{\mathtt{V}} \rightarrow \mathtt{T} & \overline{\mathtt{U}} \lesssim \overline{\mathtt{V}}\end{array}}{\Gamma \vdash_{\mathrm{G}} \mathtt{e}_0\mathtt{.m(}\overline{\mathtt{e}}\mathtt{)} : \mathtt{T}} \qquad \frac{\Gamma \vdash_{\mathrm{G}} \mathtt{e}_0 : \mathtt{?} \qquad \Gamma \vdash_{\mathrm{G}} \overline{\mathtt{e}} : \overline{\mathtt{U}}}{\Gamma \vdash_{\mathrm{G}} \mathtt{e}_0\mathtt{.m(}\overline{\mathtt{e}}\mathtt{)} : \mathtt{?}}$$

The first rule is usual except that a new relation $\lesssim$ (called consistent-subtyping after Siek and Taha [3]) is used for ordinary subtyping $<:$. This new relation takes into account the dynamic type and is defined by: $\mathtt{T} \lesssim \mathtt{U}$ if and only if either (1) one of the two types is `?` or (2) both types are class names and $\mathtt{T} <: \mathtt{U}$. Notice that $\lesssim$ is *not* (nor should it be—see [3]) a transitive relation. Other typing rules are mostly straightforward adaptations of those from FJ. We write $\vdash_{\mathrm{G}} (\mathit{CT}, \mathtt{e}) : \mathtt{T}$ to mean a program $(\mathit{CT}, \mathtt{e})$ is well typed with expression `e` being of type `T`.

The syntax of $\mathrm{FJ}_{\mathrm{refl}}$ is the same as FJ except that it has reflective member accesses `get(e, f)` and $\mathtt{invoke(e, m,} \overline{\mathtt{e}}\mathtt{)}$. The judgment for translation from $\mathrm{FJ}^?$ to $\mathrm{FJ}_{\mathrm{refl}}$ is of the form $\Gamma \vdash \mathtt{e} \rightsquigarrow \mathtt{e}' : \mathtt{T}$, read "$\mathrm{FJ}^?$ expression `e` of type `T` under environment Γ translates to $\mathrm{FJ}_{\mathrm{refl}}$ expression $\mathtt{e}'$." The translation is directed by type derivations. We show only the rules for method invocations:

[2] In this case, this error could have been detected at the same time as the method invocation. In general, however, the check for actual arguments cannot be performed at the same time, since $\mathrm{FJ}_{\mathrm{refl}}$ is not a call-by-value language (like FJ). The semantics of `invoke()` is given in such a way that a run-time check is inserted for every actual argument.

Definition 2 (Translation).

$$\frac{\Gamma \vdash \mathtt{e}_0 \rightsquigarrow \mathtt{e}_0' : \mathtt{C}_0 \qquad \Gamma \vdash \overline{\mathtt{e}} \rightsquigarrow \overline{\mathtt{e}}' : \overline{\mathtt{U}} \qquad \mathit{mtype}(\mathtt{m}, \mathtt{C}_0) = \overline{\mathtt{V}} \to \mathtt{T} \qquad \overline{\mathtt{U}} \lesssim \overline{\mathtt{V}}}{\Gamma \vdash \mathtt{e}_0\mathtt{.m(}\overline{\mathtt{e}}\mathtt{)} \rightsquigarrow \mathtt{e}_0'\mathtt{.m(}\langle\!\langle \overline{\mathtt{V}} \Leftarrow \overline{\mathtt{U}} \rangle\!\rangle \overline{\mathtt{e}}'\mathtt{)} : \mathtt{T}} \qquad \frac{\Gamma \vdash \mathtt{e}_0 \rightsquigarrow \mathtt{e}_0' : \mathtt{?} \qquad \Gamma \vdash \overline{\mathtt{e}} \rightsquigarrow \overline{\mathtt{e}}' : \overline{\mathtt{U}}}{\Gamma \vdash \mathtt{e}_0\mathtt{.m(}\overline{\mathtt{e}}\mathtt{)} \rightsquigarrow \mathtt{invoke(e}_0'\mathtt{, m, }\overline{\mathtt{e}}'\mathtt{)} : \mathtt{?}}$$

$$\langle\!\langle \mathtt{V} \Leftarrow \mathtt{U} \rangle\!\rangle \mathtt{e} \overset{\text{def}}{=} \begin{cases} \mathtt{(C)e} & (\text{if } \mathtt{V} = \mathtt{C} \text{ and } \mathtt{U} = \mathtt{?}) \\ \mathtt{e} & (\text{otherwise}) \end{cases}$$

The first rule is for ordinary invocations; $\langle\!\langle \mathtt{V} \Leftarrow \mathtt{U} \rangle\!\rangle$ inserts a run-time check when `U` is the dynamic type. The second rule deals with the case where the receiver type is `?`; the method invocation is translated into `invoke()`. The translation of type `T` (written $|\mathtt{T}|$) is obtained by replacing `?` with `Object` and that of class table CT (written $|CT|$) is by replacing every type and method body with their translations.

The form of typing judgments of expressions in $\mathrm{FJ}_{\mathrm{refl}}$ is $\Gamma \vdash_{\mathrm{R}} \mathtt{e} : \mathtt{T}$ and we use $\vdash_{\mathrm{R}} (CT, \mathtt{e}) : \mathtt{T}$ for typing of a program. We omit typing rules, since they are straightforward. Reduction $\longrightarrow_{\mathrm{R}}$ in $\mathrm{FJ}_{\mathrm{refl}}$ is basically the same as in FJ (i.e., full reduction) except the additional rules below for reflective member accesses.

Definition 3 (Reduction).

$$\frac{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}}{\mathtt{get(new\ C(}\overline{\mathtt{e}}\mathtt{), f}_i\mathtt{)} \longrightarrow_{\mathrm{R}} \mathtt{e}_i} \qquad \frac{\mathit{mbody}(\mathtt{m}, \mathtt{C}) = \overline{\mathtt{x}}.\mathtt{e}_0 \qquad \mathit{mtype}(\mathtt{m}, \mathtt{C}) = \overline{\mathtt{V}} \to \mathtt{T}}{\mathtt{invoke(new\ C(}\overline{\mathtt{e}}\mathtt{), m, }\overline{\mathtt{d}}\mathtt{)} \longrightarrow_{\mathrm{R}} [(\overline{\mathtt{V}})\overline{\mathtt{d}}/\overline{\mathtt{x}}, \mathtt{new\ C(}\overline{\mathtt{e}}\mathtt{)/this}]\mathtt{e}_0}$$

2.3 Properties

Soundness of $\mathrm{FJ}^{?}$ consists of the following two theorems, which we have proved: the first states type soundness for the intermediate language $\mathrm{FJ}_{\mathrm{refl}}$ and is proved via subject reduction and progress. The second states that a well-typed $\mathrm{FJ}^{?}$ program translates into a well-typed $\mathrm{FJ}_{\mathrm{refl}}$ program. Combining these two, we can see that a member access whose receiver is of class type is translated to an ordinary, non-reflective access, which will not fail. In other words, a statically typed portion without `?` will never fail.

Theorem 1 (Type safety of $\mathbf{FJ}_{\mathrm{refl}}$). If a program $(CT, \mathtt{e})$ satisfies $\vdash_{\mathrm{R}} (CT, \mathtt{e}) : \mathtt{T}$ and there exists a normal form expression $\mathtt{e}'$ where $\mathtt{e} \longrightarrow_{\mathrm{R}}^{*} \mathtt{e}'$, then one of the following statements holds.

1. $\mathtt{e}'$ is a value `v` where $\emptyset \vdash_{\mathrm{R}} \mathtt{v} : \mathtt{C}$ and $\mathtt{C} <: \mathtt{T}$.
2. A subexpression $\mathtt{e}_0$ in $\mathtt{e}'$ satisfies $\exists \mathtt{C}, \mathtt{D}, \overline{\mathtt{e}}.(\mathtt{e}_0 = \mathtt{(D)new\ C(}\overline{\mathtt{e}}\mathtt{)} \wedge \mathtt{C} \not<: \mathtt{D})$.
3. A subexpression $\mathtt{e}_0$ in $\mathtt{e}'$ satisfies $\exists \mathtt{C}, \overline{\mathtt{e}}, \mathtt{f}.(\mathtt{e}_0 = \mathtt{get(new\ C(}\overline{\mathtt{e}}\mathtt{), f)} \wedge \forall \overline{\mathtt{T}}, \overline{\mathtt{f}}.(\mathit{fields}(\mathtt{C}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \Rightarrow \mathtt{f} \notin \overline{\mathtt{f}}))$.
4. A subexpression $\mathtt{e}_0$ in $\mathtt{e}'$ satisfies $\exists \mathtt{C}, \overline{\mathtt{e}}, \mathtt{m}, \overline{\mathtt{d}}.(\mathtt{e} = \mathtt{invoke(new\ C(}\overline{\mathtt{e}}\mathtt{), m, }\overline{\mathtt{d}}\mathtt{)} \wedge \forall \overline{\mathtt{x}}, \mathtt{e}_0.(\mathit{mbody}(\mathtt{m}, \mathtt{C}) = \overline{\mathtt{x}}.\mathtt{e}_0 \Rightarrow \#(\overline{\mathtt{x}}) \neq \#(\overline{\mathtt{d}})))$.

Theorem 2 (Type safety of FJ$^?$). If $\vdash_{\mathrm{G}} (CT, \mathtt{e}) : \mathtt{T}$, then there exists a translation from $(CT, \mathtt{e})$ into $(|CT|, \mathtt{e}')$ satisfying $\vdash_{\mathrm{R}} (|CT|, \mathtt{e}') : |\mathtt{T}|$.

Another interesting property we have proved (but other previous work has not) is that, if an FJ$^?$ program can evolve to an FJ program (without `?`) only by changing type declarations, the FJ$^?$ program yields the same result as the evolved program, without failures at run-time checks. In the statement below, the subscript FJ means that the judgment is derived with the rules of FJ.

Theorem 3 (Safety of potentially typeable programs). Assume that $\vdash_{\mathrm{G}} (CT_1, \mathtt{e}) : \mathtt{T}_1$ and $(CT_1, \mathtt{e})$ is translated into $(|CT_1|, \mathtt{e}') \in \mathrm{FJ}_{\mathrm{refl}}$. If $\vdash_{\mathrm{FJ}} (CT_2, \mathtt{e}) : \mathtt{T}_2$ and CT_2 is obtained by replacing every `?` in CT_1 with a class name, then $\mathtt{e} \longrightarrow^*_{\mathrm{FJ}} \mathtt{v}$ under CT_2 implies $\mathtt{e}' \longrightarrow^*_{\mathrm{R}} \mathtt{v}$ under $|CT_1|$.

3 Combining Gradual Typing and Generics

In this section, we informally discuss design questions raised when combining gradual typing and generics. As we introduce generics, the syntax of types has a structure: a type is not a simple name, rather a combination of a class name as a type constructor and an appropriate number of types as arguments to it. So, the first (perhaps rather obvious) question is "Can the dynamic type `?` be a type argument?"

Our answer is definitely yes, since it will allow more flexible evolution, in particular, from a non-generic class to its generic version. For example, when class `List` is evolved to a generic version `List<X>`, the occurrences of `List` in old client code is not a type any longer, breaking the client code, but, if we think `List` (lacking a type argument) as an abbreviation of `List<?>`, there is more chance for the client code to remain well typed without modification. Indeed, the need for such an evolution was the main motivation for raw types [17, 18] in GJ (and Java 5.0). Our gradual typing for generics can be considered an enhancement of raw types with some safety guarantee[3] (the formal proof of which we leave for future work).

In what follows, we discuss two questions raised when we allow `?` to be a type argument. One is concerned about the interaction between `?` and the upper bounds of type variables. The other is about the meaning of an object whose run-time type involves `?`.

3.1 Bounded Dynamic Types

To discuss the interaction between the upper bounds of type variables and the dynamic type, first consider the following class `C` and type expression `C<?>`.

[3] For possibly unsafe use of raw types, the compiler issues an unchecked warning but there is no guarantee that the code portion without raw types is safe.

```
class C<X ◁ Number> ◁ Object {
  X f;
  int m(X x){ return x.intValue(); }
}
```

Since a method signature is supposed to be obtained by substituting actual type arguments for corresponding type variables, it seems that the invocation of `m` on `C<?>` can take `?`, namely *anything*. However, this interpretation is obviously wrong, since passing, say, a `String` to `m` would make the program crash at the invocation of `intValue()`. In other words, instantiating with `?` a type variable that is given a non-trivial upper bound will break the promise of gradual typing that a statically typed portion without `?` will never fail.

One possible remedy would be to weaken the promise and to say "A statically typed portion will never fail *as long as type variables are instantiated with types without* `?`." This sounds like a reasonable option since it is the client who abuses the class `C` by "polluting" statically typed code with `?`.

However, we do not think it is really a good idea mainly from the implementation point of view. In this case, the method `m` would have to behave in two different ways, according to what `X` is bound to: on the one hand, if `X` is instantiated with an ordinary type such as `Integer`, `m` works as usual; on the other hand, if `X` is instantiated with `?`, the invocation of `intValue()` will involve a reflective method call and run-time checks. Such an implementation will not be easy.

Another remedy, which we propose here, is to prevent non-`Number` objects from flowing into `x` *both statically and dynamically*. By "statically," we mean that an expression of an unrelated type, say `String`, as an argument to `m` is statically rejected by the compiler. By "dynamically," we mean that an expression of dynamic type as an argument to `m` is accepted by the compiler but it is checked at run time whether the value is really a `Number`.

To formally express the idea above in the type system, we introduce the notion of *bounded dynamic types*. A bounded dynamic type, written $?^{\mathtt{N}}$ where `N` is a parameterized type (or a non-variable type in the FGJ terminology), is similar to the dynamic type `?`, which is now an abbreviation of $?^{\mathtt{Object}}$, in the following sense. The type system allows (1) members that do not exist in class `N` to be accessed and (2) an expression of type $?^{\mathtt{N}}$ to be passed where a *subtype* of `N` is expected. For example, the following statements are well typed.

```
new C<?^Number>(new Float(1.0)).f.isNaN();      // (1)
Double d = new C<?^Number>(new Float(1.0)).f;  // (2)
```

In these statements, `new C<`$?^{\mathtt{Number}}$`>(...).f` has type $?^{\mathtt{Number}}$. In the first statement, method `isNaN()`, which does not exist in class `Number`, is called. In the second statement, an expression of type $?^{\mathtt{Number}}$ is passed where `Double`, which is a subtype of `Number`, is expected. It differs from `?`, however, in that (1) the usual typing rule is applied when a method existing in `N` is invoked, (2) an expression of type $?^{\mathtt{N}}$ cannot be passed to where an unrelated type is expected, and (3) a variable (or parameter) of type `?`$^{\mathsf{N}}$ cannot accept an expression of an unrelated type. For example, consider the following statements.

```
int i = new C<?^Number>(new Long(1)).f.intValue(); // (1) well typed
String s = new C<?^Number>(new Integer(1)).f;      // (2) rejected
new C<?^Number>(new Integer(1)).m("foo");          // (3) rejected
```

Again, `new C<?`$^{\mathsf{Number}}$`>(...).f` has type `?`$^{\mathsf{Number}}$. The first statement is well typed because `Number.intValue()` exists with the return type `int`. The second statement is rejected because `String` and `Number` are not related (that is, neither is a subtype of the other). The third expression is also rejected because `"foo"` has type `String` but it is not related to the expected type `Number`.

We use $\lesssim$ to perform type checking mentioned above in a way similar to FJ$^?$. It differs from FJ$^?$ in that a type has a structure and it may involve bounded dynamic types. Intuitively, `S` $\lesssim$ `T` means that replacing each `?` in `S`, `T` with an appropriate type (without `?`) below its bound will satisfy `S` $<:$ `T`. The formal definition of $\lesssim$ in this meaning is described in Section 4.

By the introduction of bounded dynamic types, we do not have to weaken the above-mentioned promise of gradual typing, since erroneous values cannot be flown into a statically typed part even when a type variable is instantiated by a (bounded) dynamic type. Also, implementation is not hard since the behavior of a statically typed part does not depend on how type variables are instantiated.

We do not expect that programmers write upper bounds for occurrences of `?`. Rather, it is the compiler's job to infer them: for each `?`, its upper bound will be the same as that of the corresponding type parameter.[4] Nevertheless, it may be a good idea to let programmers write upper bounds (and, possibly, even lower bounds as in wildcards) when they want to.

3.2 Bounded Dynamic Types and Run-Time Checks

For run-time checking, expressions are translated into the language with reflective member accesses and cast operations for explicit run-time checks in the same way as in FJ$^?$. However, we have to be careful with the cast operation since both source and target types of the cast may involve `?` like `C<?`$^{\mathsf{N}}$`>`.

We first consider the case where the target type includes `?`:

$$\texttt{C<?}^{\mathsf{Number}}\texttt{> c = e} \quad \rightsquigarrow \quad \texttt{C<?}^{\mathsf{Number}}\texttt{> c = }\langle\!\langle \texttt{C<?}^{\mathsf{Number}}\texttt{>} \rangle\!\rangle\,\texttt{e}$$

The expression assigning `e` to the variable of type `C<?`$^{\mathsf{Number}}$`>` translates into the expression with the cast operation. It states that `e` must be reduced to a value of type `C` with its type argument being a subtype of `Number`. So, when `e` is reduced, for example, to `new C<Integer>(...)`, the cast operation should succeed, on the other hand, when `e` is reduced, for example, to `new C<Object>(...)`, the operation should fail.

For the case where the source type includes `?`, consider the following example.

[4] Strictly speaking, if a type variable has an F-bound—a bound that refers to the type variable recursively, it is not clear what should be attached. We leave this issue for future work and F-bounds are omitted from the current formalization.

```
C<?^Number> c1 = new C<?^Number>(n);
C<Integer> c2 = c1;
```

where `n` is given type `Number`. Then, the variable reference `c1` in the second statement will be translated to $\langle\!\langle$`C<Integer>`$\rangle\!\rangle$ `c1`. When does this cast succeed? We conclude that it always fails no matter what `n` is: if `n` is not an `Integer`, `c2` will point to an invalid value. Another, possibly interesting option may be to allow this check to succeed when `n` happens to be an `Integer`; however, it is not clear how to achieve safety with this option—for example, when a `Float` is assigned to `c1.f`, which is of type $\texttt{?}^{\texttt{Number}}$, it becomes an invalid value for `c2.f`, an alias of `c1.f` of type `Integer`.

The run-time check required by the cast expression $\langle\!\langle \mathtt{T}_2 \rangle\!\rangle\,\mathtt{new\ C<T_1>(...)}$ can be implemented by checking $|\mathtt{C<T_1>}| \lesssim \mathtt{T}_2$, where $|\mathtt{T}|$ denotes replacing every `?` in `T` with its bound. Here, `?` in the source type needs to be replaced to implement checks as in the second case above correctly. Without $|\cdot|$, $\langle\!\langle$`C<Integer>`$\rangle\!\rangle$ `c1` would succeed since $\mathtt{C<?^{Number}>} \lesssim \mathtt{C<Integer>}$ but $|\mathtt{C<?^{Number}>}| = \mathtt{C<Number>} \not\lesssim \mathtt{C<Integer>}$.

4 Formalization

In this section, we sketch a formalization of our design discussed in the previous section. We formalize FGJ$^?$ as an extension of FGJ and FJ$^?$. For simplicity, we omit some features found in FGJ. They include F-bounded polymorphism, polymorphic methods, and typecasts, which would be easy to add. (F-bounded polymorphism makes the *inference* of appropriate bounds for `?` non-trivial, though.) The work is still in progress and we focus on typing of the surface language here.

We follow the metavariable convention of FGJ [6]. The syntax of FGJ$^?$ is mostly the same as FGJ except bounded dynamic types and the omitted features.

Definition 4 (Types and syntax of FGJ$^?$).

$$\begin{array}{ll}
\mathtt{S,T,U,V} ::= \mathtt{X} \mid \mathtt{N} \mid \mathtt{?}^{\mathtt{N}} & \mathtt{L} ::= \mathtt{class\ C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>} \triangleleft \mathtt{N}\ \{\overline{\mathtt{T}}\ \overline{\mathtt{f}}\mathtt{;}\ \mathtt{K}\ \overline{\mathtt{M}}\} \\
\mathtt{N} ::= \mathtt{C<}\overline{\mathtt{T}}\mathtt{>} & \mathtt{K} ::= \mathtt{C(}\overline{\mathtt{T}}\ \overline{\mathtt{f}}\mathtt{)\{super(}\overline{\mathtt{f}}\mathtt{);\ this.}\overline{\mathtt{f}}\mathtt{=}\overline{\mathtt{f}}\mathtt{;\}} \\
\mathtt{e} ::= \mathtt{x} \mid \mathtt{e.f} \mid \mathtt{e.m(}\overline{\mathtt{e}}\mathtt{)} \mid \mathtt{new\ N(}\overline{\mathtt{e}}\mathtt{)} & \mathtt{M} ::= \mathtt{T\ m(}\overline{\mathtt{T}}\ \overline{\mathtt{x}}\mathtt{)\{\ return\ e;\ \}}
\end{array}$$

The bound for a bounded dynamic type must be a nonvariable type `N`.

The main judgments used in the type system are: (1) $\Delta \vdash \mathtt{S} <: \mathtt{T}$ for subtyping; (2) $\Delta \vdash \mathtt{S} \lesssim \mathtt{T}$ for consistent-subtyping; (3) $\Delta \vdash \mathtt{T}$ ok for well-formed types; and (4) $\Delta; \Gamma \vdash \mathtt{e} : \mathtt{T}$ for expression typing, where Δ is a bound environment, which is a finite mapping from type variables to nonvariable types.

The subtyping relation is basically the same as that in FGJ. Note that bounded dynamic types are related only to themselves by reflexivity. When actual and formal argument types are compared, we use the consistent-subtyping relation $\lesssim$ as in FJ$^?$. It is defined via the consistency relation $\sim$, which relates dynamic types and ordinary types:

Definition 5 (Consistent-subtyping).

$$\frac{\Delta \vdash \overline{\mathtt{S}} \sim \overline{\mathtt{T}}}{\Delta \vdash \mathtt{C<}\overline{\mathtt{S}}\mathtt{>} \sim \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}} \qquad \frac{\Delta \vdash \mathtt{T} \sim \mathtt{N}}{\Delta \vdash \mathtt{T} \sim \mathtt{?}^{\mathtt{N}}} \qquad \frac{\Delta \vdash \mathtt{T} <: \mathtt{N}}{\Delta \vdash \mathtt{T} \sim \mathtt{?}^{\mathtt{N}}} \qquad \frac{\Delta \vdash \mathtt{S} <: \mathtt{T} \qquad \Delta \vdash \mathtt{T} \sim \mathtt{U}}{\Delta \vdash \mathtt{S} \lesssim \mathtt{U}}$$

$\sim$ is reflexive and symmetric (we omit rules for these). As seen from the last rule, the rules are not algorithmic. The development of a decision algorithm for $\lesssim$ is not yet finished.

Typing rules are actually obtained by combining those of FJ$^?$ and of FGJ in a straightforward manner. Here are the two rules of method invocation.

Definition 6 (Typing rules of expressions).

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash_{\mathrm{G}} \mathtt{e}_0 : \mathtt{T}_0 \quad \Delta; \Gamma \vdash_{\mathrm{G}} \overline{\mathtt{e}} : \overline{\mathtt{V}} \\ \mathit{mtype}(\mathtt{m}, \mathit{bound}_\Delta(\mathtt{T}_0)) = \overline{\mathtt{U}} \rightarrow \mathtt{U} \quad \Delta \vdash \overline{\mathtt{V}} \lesssim \overline{\mathtt{U}}\end{array}}{\Delta; \Gamma \vdash_{\mathrm{G}} \mathtt{e}_0\mathtt{.m(}\overline{\mathtt{e}}\mathtt{)} : \mathtt{U}} \qquad \frac{\begin{array}{c}\Delta; \Gamma \vdash_{\mathrm{G}} \mathtt{e}_0 : \mathtt{?}^{\mathtt{N}} \quad \Delta; \Gamma \vdash_{\mathrm{G}} \overline{\mathtt{e}} : \overline{\mathtt{V}} \\ \mathit{nomethod}(\mathtt{m}, \mathit{bound}_\Delta(\mathtt{?}^{\mathtt{N}}))\end{array}}{\Delta; \Gamma \vdash_{\mathrm{G}} \mathtt{e}_0\mathtt{.m(}\overline{\mathtt{e}}\mathtt{)} : \mathtt{?}^{\mathtt{Object}}}$$

The first rule is the typing rule of method invocation when the method definitely exists, i.e., $\mathit{mtype}(\mathtt{m}, \mathit{bound}_\Delta(\mathtt{T}_0)) = \overline{\mathtt{U}} \rightarrow \mathtt{U}$, in the receiver. We define $\mathit{bound}_\Delta(\mathtt{?}^{\mathtt{N}}) = \mathit{bound}_\Delta(\mathtt{N})$, which means that, even if the receiver type is dynamic, the number of arguments and actual argument types are checked statically (by using $\lesssim$). The second rule is the other typing rule of method invocation, applied when the receiver type is dynamic and the method may not exist. $\mathit{nomethod}(\mathtt{m}, \mathit{bound}_\Delta(\mathtt{?}^{\mathtt{T}_0}))$ means that the specified class and its super classes do not have method named `m`. Nevertheless, this method invocation is allowed statically, since, at run time, the value of $\mathtt{e}_0$ may have method named `m`. This kind of method invocation will be translated into a reflective call, which checks if method `m` really exists.

The definitions of the intermediate language and the translation to it are omitted but mostly complete along the line described in the previous section. We need to prove desired properties, including type soundness of the intermediate language, type preservation of the translation, and safety of potentially typeable programs, as we have done for FJ$^?$.

5 Conclusion

This paper reports work in progress on the introduction of gradual typing into class-based object-oriented languages with Java-style generics. We have reviewed the development of the gradual typing system FJ$^?$, which translates into a language with reflective member accesses for explicit run-time checks. Then, we have informally discussed issues raised when gradual typing and generics are combined and sketched an extension FGJ$^?$ of FJ$^?$ with generics.

The main question is how to deal with the instantiation of a type variable with the dynamic type, especially, when the type variable has a non-trivial upper bound. To address the question, we have introduced the notion of bounded dynamic types, which have characteristics of both ordinary types and the dynamic type. They allow us to avoid weakening the partial safety of gradual typing.

Immediate future work is, of course, to complete the theoretical development, including proofs of desired properties. The usefulness of bounded dynamic types in real program evolution scenarios should be examined.

Acknowledgments. This work was supported in part by Grant-in-Aid for Young Scientists (A) No. 21680002. We thank anonymous reviewers for valuable comments.

References

1. Siek, J.G., Taha, W.: Gradual typing for functional languages. In: Proc. of the Scheme and Functional Programming Workshop. (September 2006)
2. Torgersen, M.: New features in C# 4.0. `http://code.msdn.microsoft.com/csharpfuture`
3. Siek, J.G., Taha, W.: Gradual typing for objects. In: Proc. of ECOOP'07. Volume 4509 of Springer LNCS. (2007) 2–27
4. Abadi, M., Cardelli, L.: A Theory of Objects. Springer Verlag (1996)
5. Ina, L., Igarashi, A.: Gradual typing for Featherweight Java. Computer Software **26**(2) (April 2009) In Japanese.
6. Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight Java: A minimal core calculus for Java and GJ. ACM TOPLAS **23**(3) (May 2001) 396–450
7. Bracha, G., Griswold, D.: Strongtalk: Typechecking Smalltalk in a production environment. In: Proc. of OOPSLA'93. (1993) 215–230
8. Thiemann, P.: Towards a type system for analyzing JavaScript programs. In: Proc. of ESOP'09. Volume 3444 of Springer LNCS. (2005) 408–422
9. Furr, M., An, J., Foster, J.S., Hicks, M.: Static type inference for ruby. In: Proc. of ACM Symposium on Applied Computing (SAC'09). (March 2009) 1859–1866
10. Anderson, C., Drossopoulou, S.: BabyJ - from object based to class based programming via types. In: Proc. of WOOD'03. Volume 82 of ENTCS. (2003)
11. Lagorio, G., Zucca, E.: Just: safe unknown types in Java-like languages. Journal of Object Technology **6**(2) (February 2007) 69–98
12. Gray, K.E., Findler, R.B., Flatt, M.: Fine-grained interoperability through mirrors and contracts. In: Proc. of ACM OOPSLA'05. (2005) 231–245
13. Findler, R.B., Felleisen, M.: Contracts for higher-order functions. In: Proc. of ACM ICFP'02. (2002) 48–59
14. Siek, J.G., Vachharajani, M.: Gradual typing with unification-based inference. In: Proc. of Dynamic Language Symposium (DLS'08). (July 2008)
15. Matthews, J., Ahmed, A.: Parametric polymorphism throught run-time sealing, or, thorems for low, low prices! In: Proc. of ESOP'08. Volume 4960 of Springer LNCS. (2008) 16–31
16. Torgersen, M., Hansen, C.P., Ernst, E., von der Ahé, P., Bracha, G., Gafter, N.: Adding wildcards to the Java programming language. In: Proc. of ACM Symposium on Applied Computing (SAC'04). (March 2004) 1289–1296
17. Bracha, G., Odersky, M., Stoutamire, D., Wadler, P.: Making the future safe for the past: Adding genericity to the Java programming language. In: Proc. of ACM OOPSLA'98. (1998) 183–200
18. Igarashi, A., Pierce, B.C., Wadler, P.: A recipe for raw types. In: Informal Proc. of FOOL8, London, England (January 2001) 65–82